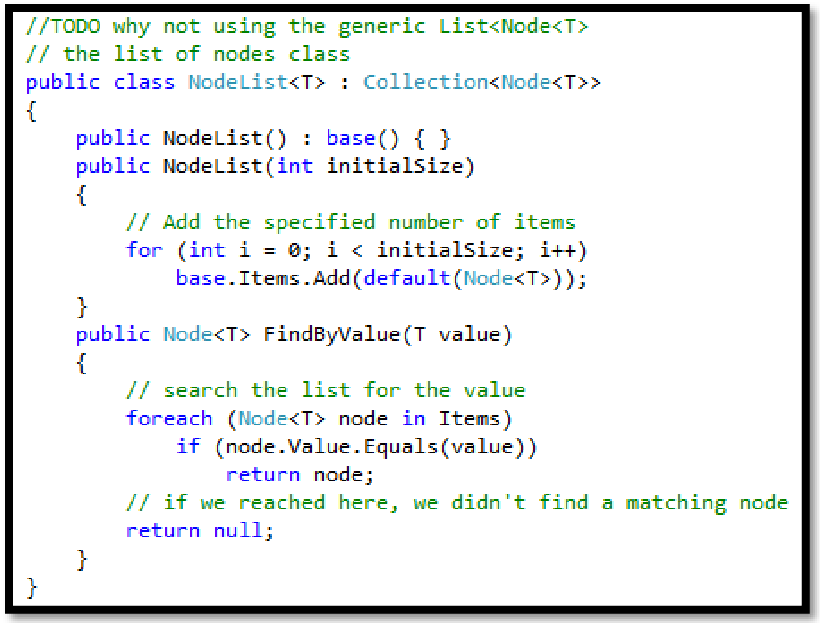
Lecture 10

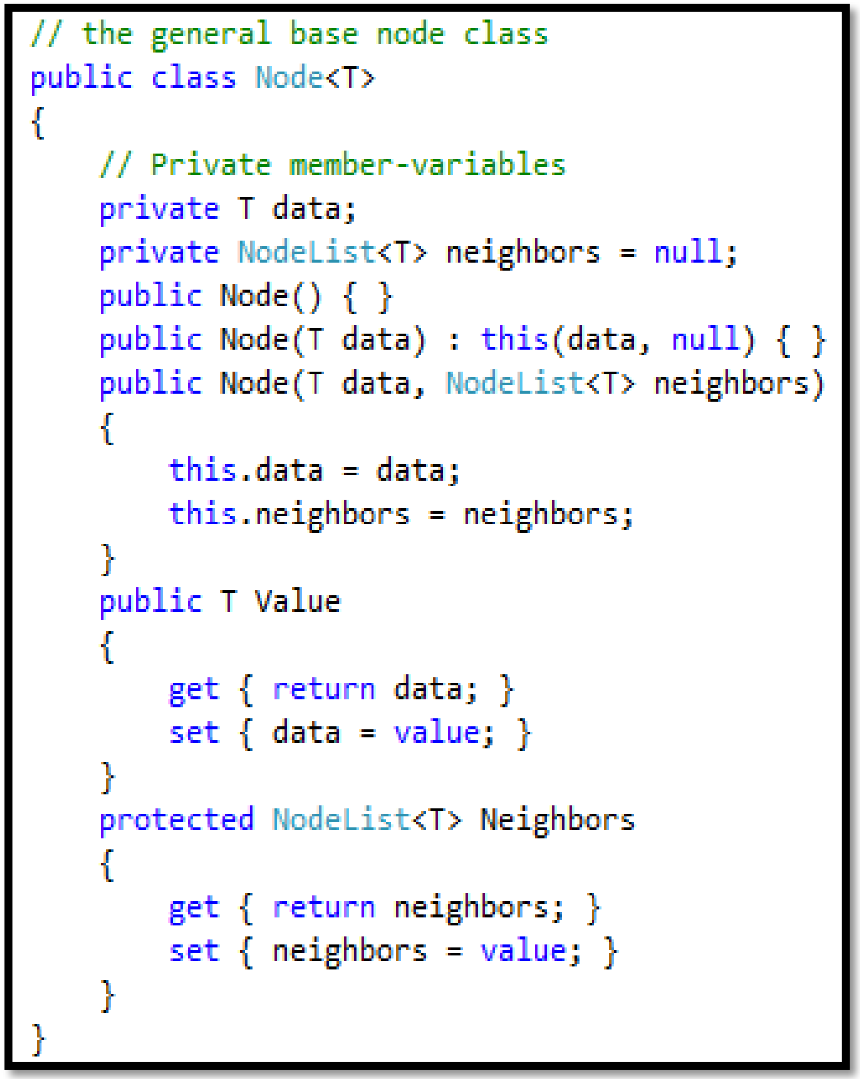
Graph implementation

The graph node class base class: Node<T> and NodeList class

Recall: this is the base node class used as the parent class for the Binary tree node class. It could be used also as the parent class for the Graph node class. The class use an instant of the NodeList class to represent the node neighbors



```
//TODO why not using the generic List<Node<T>
// the list of nodes class
public class NodeList<T> : Collection<Node<T>>
{
    public NodeList() : base() { }
    public NodeList(int initialSize)
    {
        // Add the specified number of items
        for (int i = 0; i < initialSize; i++)
            base.Items.Add(default(Node<T>));
    }
    public Node<T> FindByValue(T value)
    {
        // search the list for the value
        foreach (Node<T> node in Items)
            if (node.Value.Equals(value))
                return node;
        // if we reached here, we didn't find a matching node
        return null;
    }
}
```

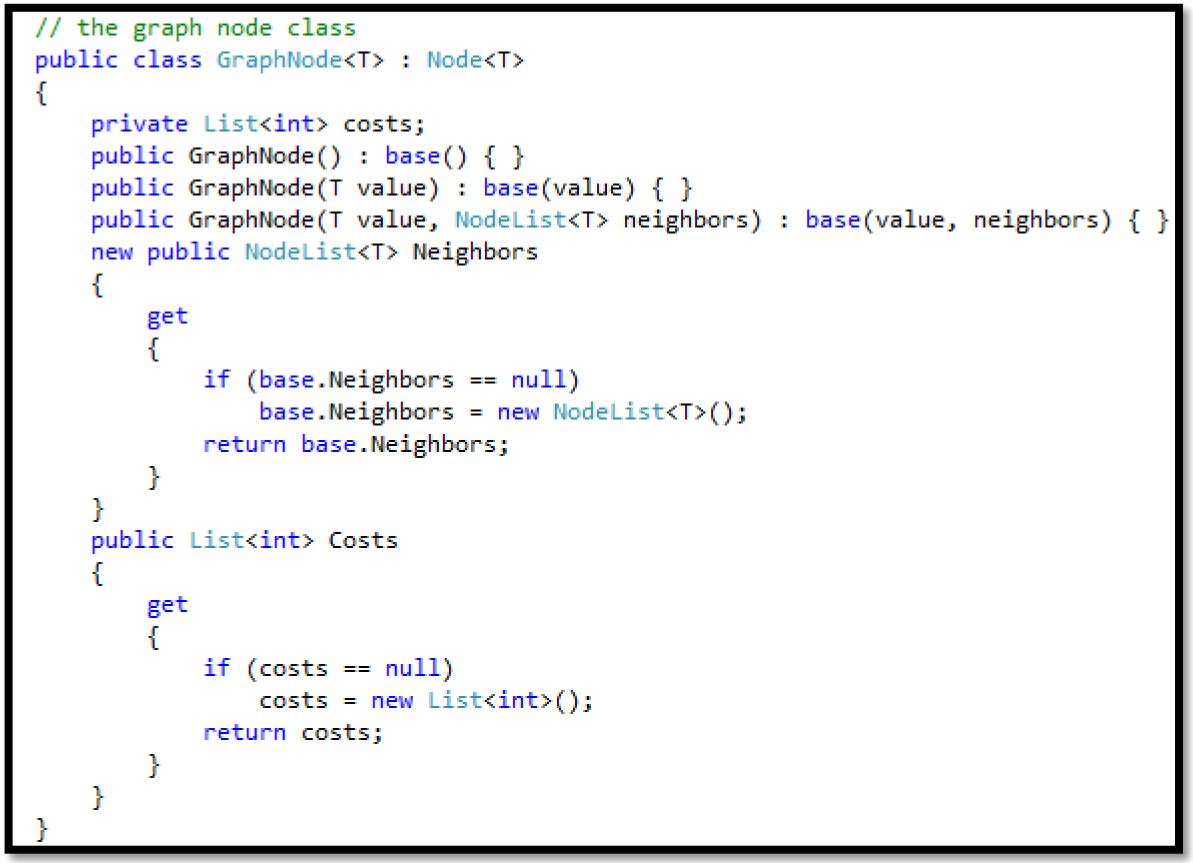


```
// the general base node class
public class Node<T>
{
    // Private member-variables
    private T data;
    private NodeList<T> neighbors = null;
    public Node() { }
    public Node(T data) : this(data, null) { }
    public Node(T data, NodeList<T> neighbors)
    {
        this.data = data;
        this.neighbors = neighbors;
    }
    public T Value
    {
        get { return data; }
        set { data = value; }
    }
    protected NodeList<T> Neighbors
    {
        get { return neighbors; }
        set { neighbors = value; }
    }
}
```

A generic List<Node<T>) could be used instead of a separated class NodeList but this requires implementing Icomparable on the Node class

The GraphNode class

The GraphNode class extentends the Node class by providing public access to the Neighbors Node list and adding the Cost to represent the weight or the cost for travelling from this node to one of its neighbors

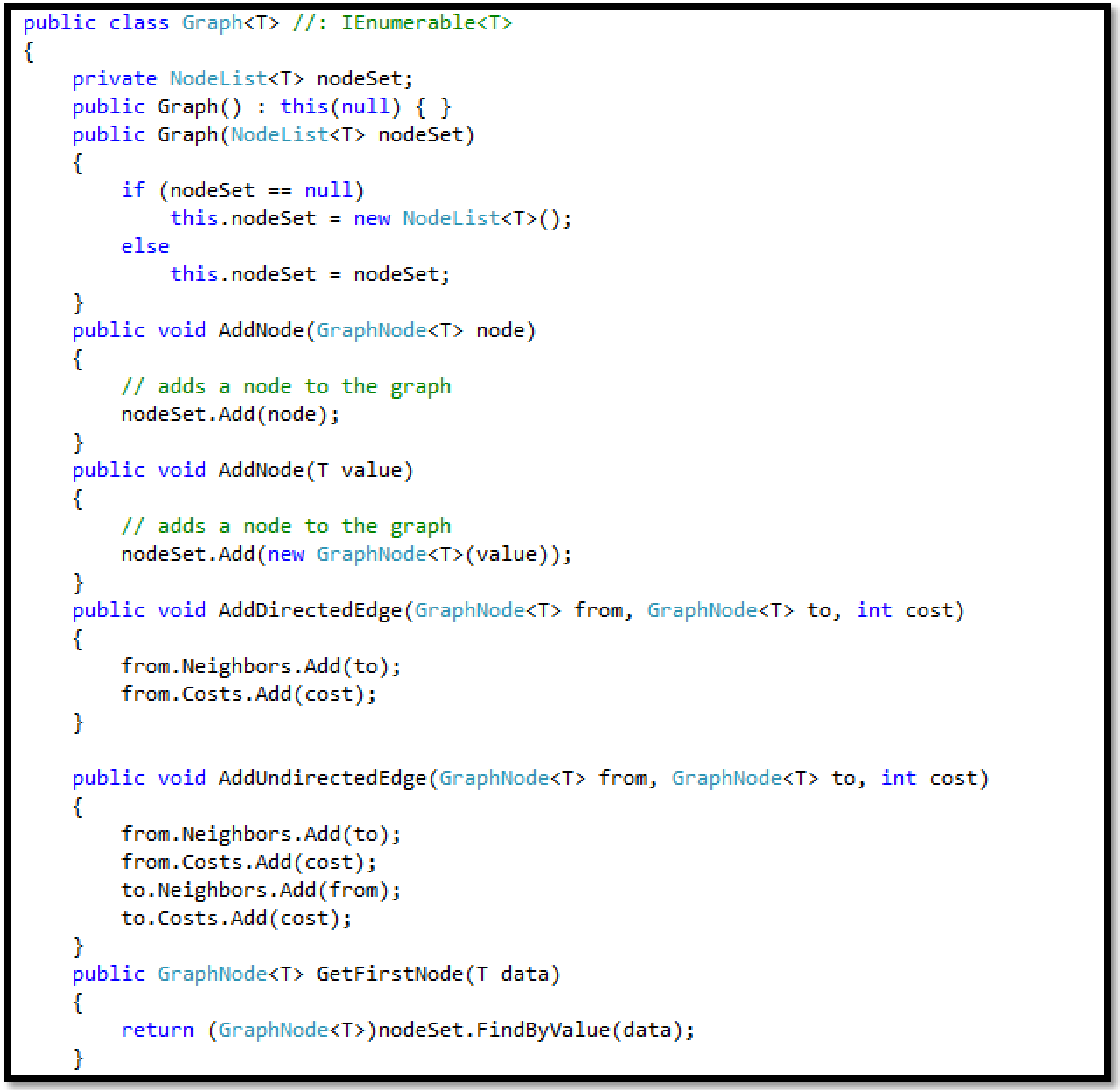


```
// the graph node class
public class GraphNode<T> : Node<T>
{
    private List<int> costs;
    public GraphNode() : base() { }
    public GraphNode(T value) : base(value) { }
    public GraphNode(T value, NodeList<T> neighbors) : base(value, neighbors) { }
    new public NodeList<T> Neighbors
    {
        get
        {
            if (base.Neighbors == null)
                base.Neighbors = new NodeList<T>();
            return base.Neighbors;
        }
    }
    public List<int> Costs
    {
        get
        {
            if (costs == null)
                costs = new List<int>();
            return costs;
        }
    }
}
```

The Graph class

- Recall that with the adjacency list technique, the graph maintains a list of its nodes. Each node, then, maintains a list of adjacent nodes. So, in creating the Graph class we need to have a list of GraphNodes. This set of nodes is maintained using a NodeList instance
- the Graph class has a number of methods for adding nodes and directed or undirected and weighted or unweighted edges between nodes. The AddNode() method adds a node to the graph, while AddDirectedEdge() and AddUndirectedEdge() allow a weighted or unweighted edge to be associated between two nodes.
- the Graph class has a Contains() method that returns a Boolean indicating if a particular value exists in the graph or not.

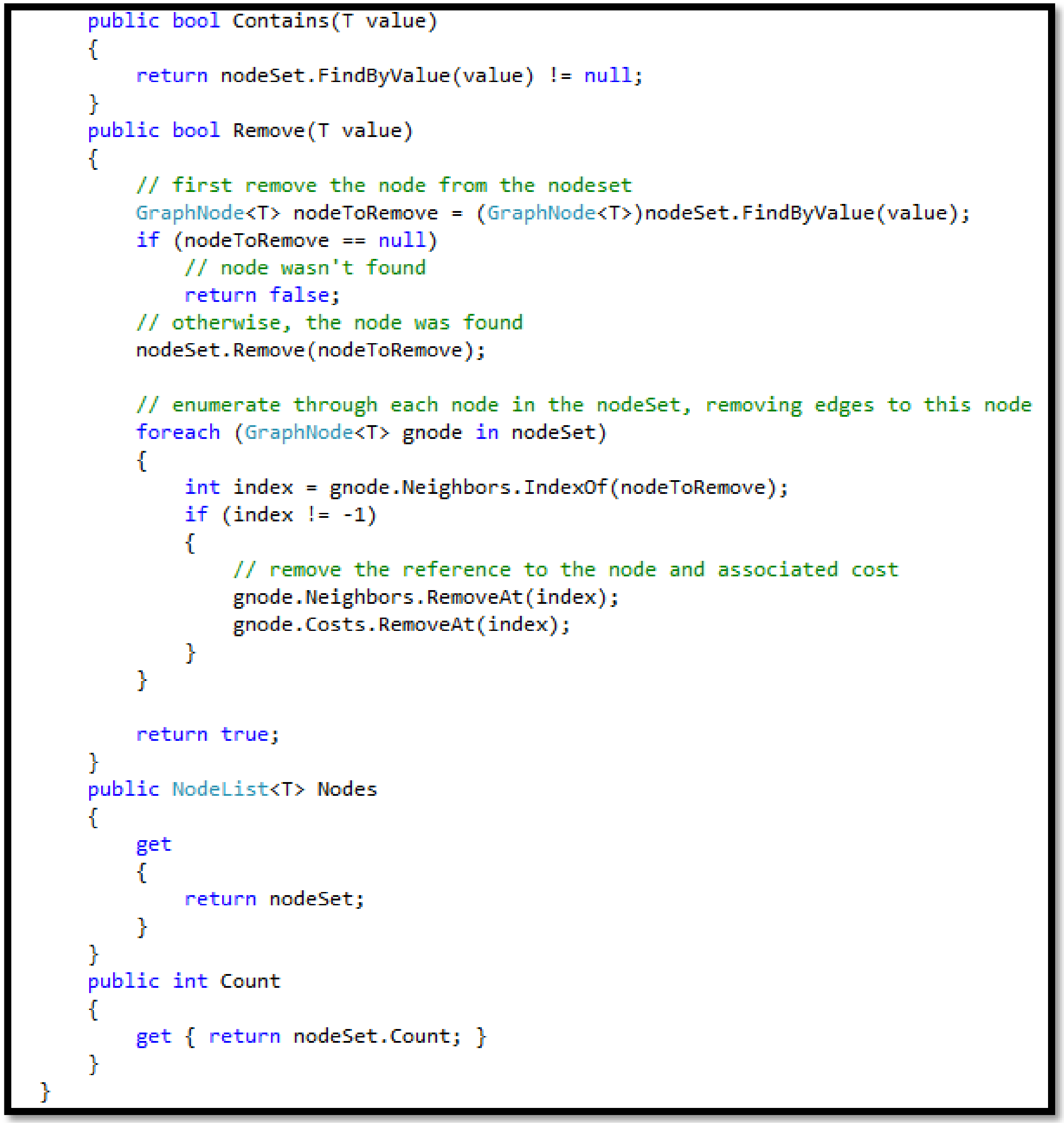
The class is continued in the next slide



```
public class Graph<T> //: IEnumerable<T>
{
    private NodeList<T> nodeSet;
    public Graph() : this(null) { }
    public Graph(NodeList<T> nodeSet)
    {
        if (nodeSet == null)
            this.nodeSet = new NodeList<T>();
        else
            this.nodeSet = nodeSet;
    }
    public void AddNode(GraphNode<T> node)
    {
        // adds a node to the graph
        nodeSet.Add(node);
    }
    public void AddNode(T value)
    {
        // adds a node to the graph
        nodeSet.Add(new GraphNode<T>(value));
    }
    public void AddDirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost)
    {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);
    }

    public void AddUndirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost)
    {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);
        to.Neighbors.Add(from);
        to.Costs.Add(cost);
    }
    public GraphNode<T> GetFirstNode(T data)
    {
        return (GraphNode<T>)nodeSet.FindByValue(data);
    }
```

The Graph class continued



```
        public bool Contains(T value)
        {
            return nodeSet.FindByValue(value) != null;
        }
        public bool Remove(T value)
        {
            // first remove the node from the nodeset
            GraphNode<T> nodeToRemove = (GraphNode<T>)nodeSet.FindByValue(value);
            if (nodeToRemove == null)
                // node wasn't found
                return false;
            // otherwise, the node was found
            nodeSet.Remove(nodeToRemove);

            // enumerate through each node in the nodeSet, removing edges to this node
            foreach (GraphNode<T> gnode in nodeSet)
            {
                int index = gnode.Neighbors.IndexOf(nodeToRemove);
                if (index != -1)
                {
                    // remove the reference to the node and associated cost
                    gnode.Neighbors.RemoveAt(index);
                    gnode.Costs.RemoveAt(index);
                }
            }

            return true;
        }
        public NodeList<T> Nodes
        {
            get
            {
                return nodeSet;
            }
        }
        public int Count
        {
            get { return nodeSet.Count; }
        }
    }
```

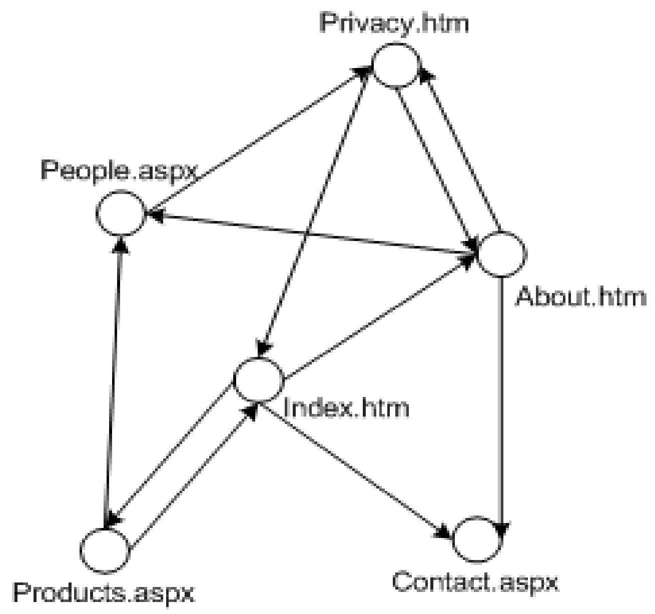
Report Discussion

Last lecture report: How could you model the delta cities map using graphs

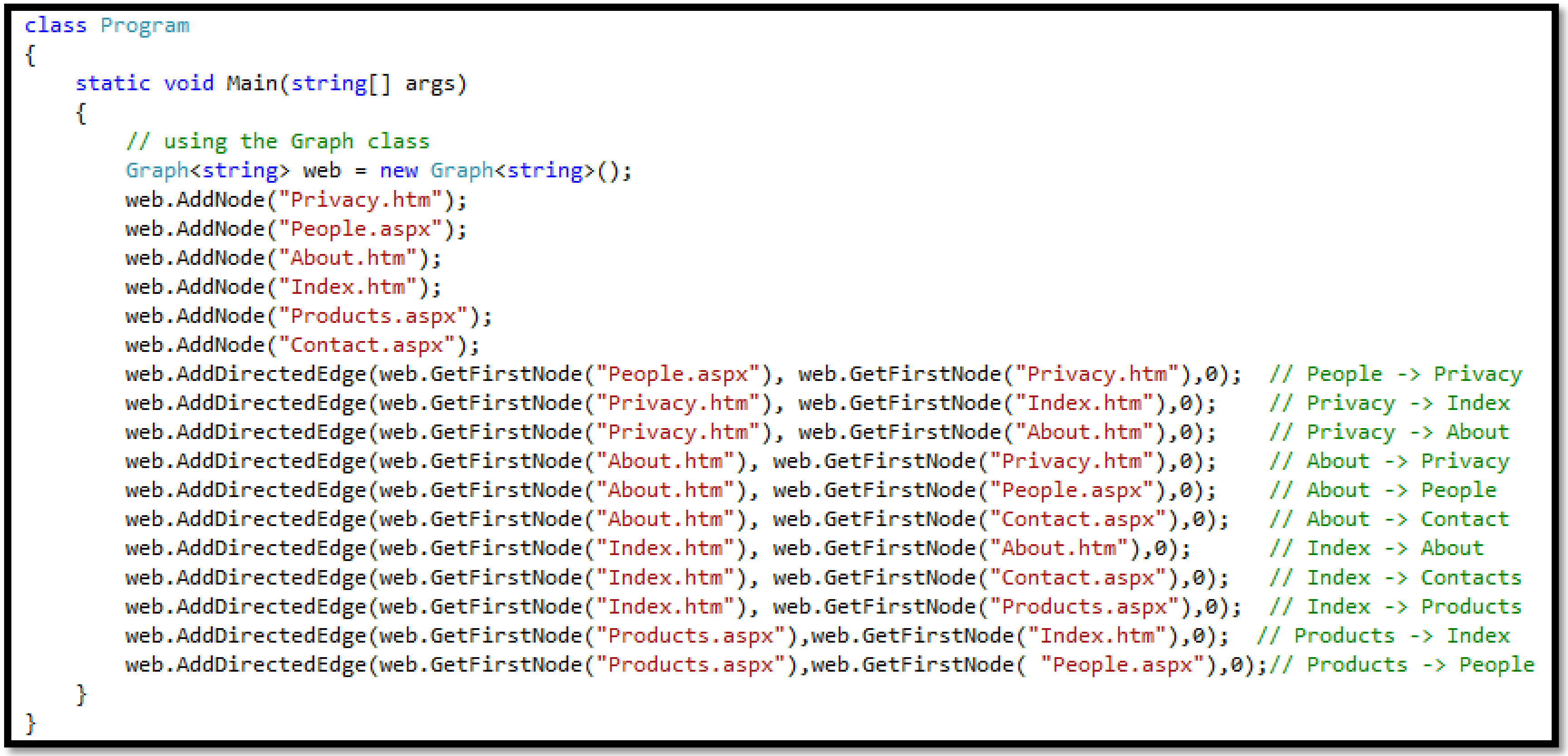
New report: Using the Graph class, implement the graph you formed in the last lecture report

Using the Graph class

Use the Graph class to implement the following graph



Using the Graph class



```
class Program
{
    static void Main(string[] args)
    {
        // using the Graph class
        Graph<string> web = new Graph<string>();
        web.AddNode("Privacy.htm");
        web.AddNode("People.aspx");
        web.AddNode("About.htm");
        web.AddNode("Index.htm");
        web.AddNode("Products.aspx");
        web.AddNode("Contact.aspx");
        web.AddDirectedEdge(web.GetFirstNode("People.aspx"), web.GetFirstNode("Privacy.htm"),0);  // People -> Privacy
        web.AddDirectedEdge(web.GetFirstNode("Privacy.htm"), web.GetFirstNode("Index.htm"),0);    // Privacy -> Index
        web.AddDirectedEdge(web.GetFirstNode("Privacy.htm"), web.GetFirstNode("About.htm"),0);    // Privacy -> About
        web.AddDirectedEdge(web.GetFirstNode("About.htm"), web.GetFirstNode("Privacy.htm"),0);    // About -> Privacy
        web.AddDirectedEdge(web.GetFirstNode("About.htm"), web.GetFirstNode("People.aspx"),0);    // About -> People
        web.AddDirectedEdge(web.GetFirstNode("About.htm"), web.GetFirstNode("Contact.aspx"),0);   // About -> Contact
        web.AddDirectedEdge(web.GetFirstNode("Index.htm"), web.GetFirstNode("About.htm"),0);      // Index -> About
        web.AddDirectedEdge(web.GetFirstNode("Index.htm"), web.GetFirstNode("Contact.aspx"),0);   // Index -> Contacts
        web.AddDirectedEdge(web.GetFirstNode("Index.htm"), web.GetFirstNode("Products.aspx"),0);  // Index -> Products
        web.AddDirectedEdge(web.GetFirstNode("Products.aspx"),web.GetFirstNode("Index.htm"),0);  // Products -> Index
        web.AddDirectedEdge(web.GetFirstNode("Products.aspx"),web.GetFirstNode( "People.aspx"),0);// Products -> People
    }
}
```